Chapter 3

Consistency-enforcing algorithms: constraint propagation

3.1 Introduction

Perhaps the most exciting and fundamental concept that drives the constraint processing area is *constraint propagation*. These are inference methods used by us in everyday life which can be imitated by computers to exhibit intelligent inference.

Assume again our party example in Chapter 2 where we would like to invite Alex, Bill, and Chris to a party. Let A, B, and C denote the propositions "Alex comes", "Bill comes" and "Chris comes," respectively. If Alex comes to the party, Bill will come as well, and if Chris comes, then Alex will too. This is expressed using Boolean constraints as $(A \rightarrow B)$ and $(C \rightarrow A)$. Assume now that Chris will definitely come to the party, namely C is a fact (a unary constraint, or a domain constraint). Using the constraint $(C \rightarrow A)$ and C, we can immediately infer A, that Alex will be at the party. Now, using the new deduced fact A and the single constraint $(A \rightarrow B)$ you can immediately conclude B, that Bill will be at the party. We can similarly reason in the reverse order. Suppose you know the two rules above and also that Bill did not go to the party after all. That is, you know $\neg B$. From $\neg B$ and $(A \rightarrow B)$ you can deduce $\neg A$. From $\neg A$ and $(C \rightarrow A)$ you can infer $\neg C$. Both chains of reasoning demonstrate *(Boolean) constraint propagation*, the simplest form of inference. Each deduction step involves just a single variable and a single constraint. The deduction is performed by *propagating* the local reasoning steps. Propagation is efficient, requiring minimal new memory and yields valuable conclusions. *Constraint propagation* is at the heart of all the more complex inference methods, which are the essence of constraint techniques.

This chapter deals with the processing and solving of constraint systems by inference. In general, inference, as it is applied to constraints, narrows the search space of possible

partial solutions by creating equivalent, yet more explicit, networks. We demonstrate the relevant concepts employing example constraints such as $x \neq y$ or $x = y$ that are easy to specify and manipulate. Consider the constraint network having three variables x, y, z, all with domains $D = \{red, blue\}$, and three constraints: (1) $x = y$, (2) $y = z$, and (3) $x \neq z$. From the first two constraints, we can infer that $x = z$. Since this inferred constraint conflicts with the constraint $x \neq z$, we can conclude that the set of three constraints is inconsistent.

Consider now the set of just two constraints $\mathcal{R} = \{x = y, y = z\}$. From $\mathcal{R}$ we can infer $x = z$, which we can add to the set of constraints, resulting in the new network $\mathcal{R}' = \{x = y, y = z, x = z\}$. The two networks $\mathcal{R}$ and $\mathcal{R}'$ are equivalent, meaning that they have the same set of solutions, but $\mathcal{R}'$ is more explicit than $\mathcal{R}$; it includes a relationship not made explicit in $\mathcal{R}$.

Suppose now that we want to generate a solution for these networks by assigning values to the variables along the ordering x, z, y. Using $\mathcal{R}$, we can generate the partial solution $(x = red, z = blue)$ and will only then realize that there is not a consistent extension to y. In other words, the two values in the domain of the remaining variable, y, are both inconsistent with at least one value assignment in the partial solution. Therefore, we have to backtrack and to try another assignment for z. This time, only $z = red$ remains, which yields a full solution.

Let's now assign values along the same ordering while consulting network $\mathcal{R}'$. Since the constraint $x = z$ is explicit in $\mathcal{R}'$, it will not allow us to assign a value to z that differs from the value assigned to x. Therefore, the partial assignment above is not a partial solution of $\mathcal{R}'$, and we will avoid the dead-end we encountered with $\mathcal{R}$.

In general, the more explicit and tight (but equivalent) constraint networks are, the more restricted the search space of all possible partial solutions will be, and consequently, any search becomes more efficient. In fact the problem may become explicit enough (by inferring additional constraints or by tightening existing ones) that the search will be directed to a solution without encountering a dead-end.

Indeed, constraint inference can be used to find a complete solution. It can deduce constraints from the given initial set until either an inconsistency is encountered, or until an equivalent constraint network is created from which every solution can be derived by a depth-first search that encounters no dead-ends.

Unfortunately solving a complete problem by inference is frequently too hard, requiring the addition of an exponential number of new constraints. A more conservative approach is to infer only a restricted set of new constraints, yielding a reduced search space. The resulting new network may not face any dead-ends, but it may allow finding a solution, or concluding inconsistency, relatively quickly. Algorithms that perform a bounded amount of constraint inference are called *local consistency enforcing*, *bounded inference*, or *constraint propagation* algorithms.

Since consistency enforcing algorithms were introduced to assist search, they are defined in terms of extending a solution of a fixed length $i-1$ by one more variable. Intuitively, a consistency-enforcing algorithm makes any partial solution of a small subnetwork extendible to some surrounding network. The primary characteristic of consistency inference algorithms is the *size* of the subnetwork involved in the inference. Size can be defined by the number of variables or by the number of constraints involved. Algorithms that infer constraints based on pairs of variables, are called *arc-consistency algorithms*. Arc-consistency ensures that any legal value in the domain of a single variable has a legal match in the domain of any other selected variable. Algorithms that infer constraints based on subnetworks of size three, are called *path-consistency*. Path-consistency ensures that any consistent solution to a two-variable subnetwork is extendible to any third variable. More generally, algorithms that infer constraints based on subnetworks having i variables, are called *i-consistency*. These algorithms guarantee that any consistent instantiation of $i-1$ variables is extendible to any i^{th} variable. If a network is i-consistent for all i we call it *globally consistent*.

How much consistency should be enforced on the network before commencing search? Though any search algorithm will benefit from representations that have a high level of consistency, the time and space cost of enforcing i-consistency is exponential in i. Thus, there is a trade-off between the effort spent on preprocessing by i-consistency (inferring constraints based on i-variables) and that spent on subsequent search. Experimental analysis of local consistency enforcement, both before and during search, focus on this trade-off.

The remainder of this chapter covers the basics of local consistency methods. We present consistency enforcing methods whose level is determined by the number of variables involved. Most of the chapter focuses on arc and path-consistency, which are inherently binary-networks concepts. Higher levels of local-consistency as well as extensions of arc-consistency to the non-binary case, are introduced towards the end. In Chapter 8 we will return to local consistency, and focus on definitions and methods based on the number of constraints involved, and apply the entire exposition to general constraints.

Before we move on, lets review the definition of a central concept in this area, *consistency of a partial solution*.

Definition 3.1.1 (partial solution) *Given a constraint network $\mathcal{R}$, we say that an assignment of values to a subset of the variables $S = \{x_1, ..., x_j\}$ given by $\bar{a} = (< x_1, a_1 > , < x_2, a_2 >, ..., < x_j, a_j >)$ is consistent relative to $\mathcal{R}$ iff it satisfies every constraint R_{S_i} such that $S_i \subseteq S$. The assignment $\bar{a}$ is also called a partial solution of $\mathcal{R}$. The set of all partial solutions of a subset of variables S is denoted by ρ_S or $\rho(S)$.*

3.2 Arc-consistency

Note that the minimal network has the following *local-consistency* property: any value in the domain of a single variable can be extended consistently by any other variable (this follows immediately from Proposition 2.3.15). This property is termed *arc-consistency*, and it can be satisfied by non-minimal networks as well. Arc-consistency can be enforced on any network by an efficient computation that, because of its local and distributed character, is often called *propagation*.

The following example more clearly demonstrates the notion of *arc-consistency*. (We speak both of a constraint being arc-consistent (or not) relative to a given variable and of a variable being arc-consistent (or not) relative to other variables. In both cases, the underlying meaning is the same.)

Example 3.2.1 Consider the variables x and y, whose domains are $D_x = D_y = \{1, 2, 3\}$, and the single constraint R_{xy} expressing the relation $x < y$. The constraint R_{xy} is depicted in a *matching diagram*[1] in Figure 3.1a, where the domain of each variable is an enclosed set of points, and arcs connect points that correspond to consistent pairs of values. (Note: this type of diagram should not be confused with the constraint graph of the network.) Because the value $3 \in D_x$ has no consistent matching value in D_y, we say that the constraint R_{xy} is not arc-consistent relative to x. Similarly, R_{xy} is not arc-consistent relative to y, since $y = 1$ has no consistent match in x. In matching diagrams, a constraint is not arc-consistent if any of its variables has *lonely* values.

Now, if we shrink the domains of both x and y such that $D_x = \{1, 2\}$ and $D_y = \{2, 3\}$, then x is arc-consistent relative to y, and y is arc-consistent relative to x. The matching diagram of the arc-consistent constraint network is depicted in Figure 3.1b. If we shrink the domains even further to $D_x = \{1\}$ and $D_y = \{2\}$, we will still have an arc-consistent constraint. However, the latter is no longer equivalent to the original constraint since we may have deleted solutions from the whole set of solutions. □

Definition 3.2.2 (arc-consistency) *Given a constraint network $\mathcal{R} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, with $R_{ij} \in C$, a variable x_i is* arc-consistent *relative to x_j if and only if for every value $a_i \in D_i$ there exists a value $a_j \in D_j$ such that $(a_i, a_j) \in R_{ij}$. The subnetwork (alternatively, the arc) defined by $\{x_i, x_j\}$ is arc-consistent if and only if x_i is arc-consistent relative to x_j and x_j is arc-consistent relative to x_i. A network of constraints is called* arc-consistent *iff all of its arcs (e.g., subnetworks of size 2) are arc-consistent.*

As we saw in the above example, we can make a binary constraint arc-consistent by shrinking the domains of the variables in its scope. If a value does not participate in a

[1] Also called micro-structure [153]

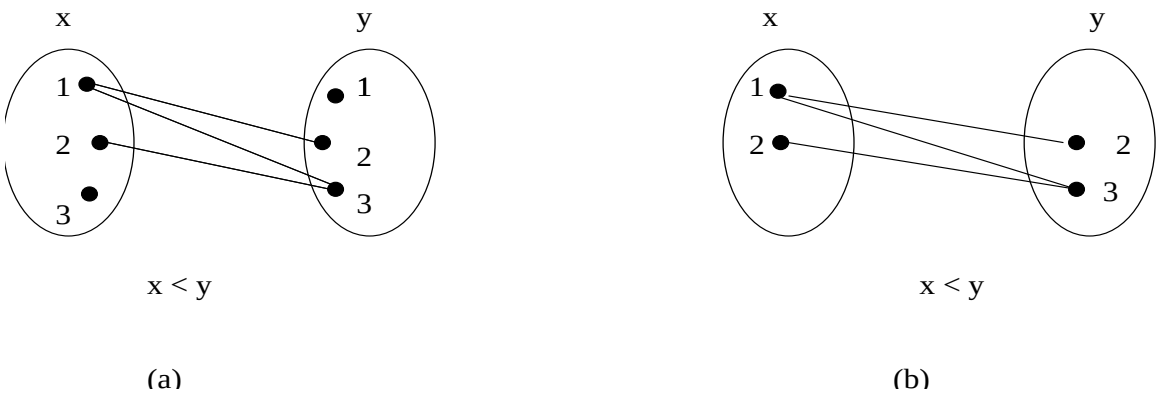

Figure 3.1: A matching diagram describing the arc-consistency of two variables x and y. In (a) the variables are not arc-consistent. In (b) the domains have been reduced, and the variables are now arc-consistent.

solution of a two-variable subnetwork, it will clearly not be part of a complete solution. But how do we ensure that we only eliminate values that will not affect the set of the network's solutions? The simple procedure *Revise*$((x_i), x_j)$, shown in Figure 3.2, if applied to two variables, x_i and x_j, returns the largest domain D_i of x_i for which x_i is arc-consistent relative to x_j. It simply tests each value of x_i and eliminates those values having no match in x_j.

REVISE$((x_i), x_j)$
input: a subnetwork defined by two variables $X = \{x_i, x_j\}$, a distinguished variable x_i, domains: D_i and D_j, and constraint R_{ij}
output: D_i, such that x_i is arc-consistent relative to x_j
1. **for** each $a_i \in D_i$
2. **if** there is no $a_j \in D_j$ such that $(a_i, a_j) \in R_{ij}$
3. **then** delete a_i from D_i
4. **endif**
5. **endfor**

Figure 3.2: The Revise procedure

Since each value in D_i is compared, in the worst case, with each value in D_j, Revise has the following complexity:

Proposition 3.2.3 *The complexity of Revise is $O(k^2)$, where k bounds the domain size.* □

Revise can also be described using composition; namely, lines 1, 2, and 3 can be replaced by

$$D_i \leftarrow D_i \cap \pi_i(R_{ij} \bowtie D_j) \tag{3.1}$$

In this case, D_i stands for the one-column relation over x_i. (consult section 1.3 for definitions of join and project operators.) Remember that the subscript i is shorthand for variable x_i.

Arc-consistency may be imposed on some pairs of variables, on all pairs from some subset of variables, or over an entire network. Arc-consistency of a whole network is accomplished by applying the Revise procedure to all pairs of variables, though, applying the procedure just once to each pair of variables is sometimes not enough to ensure the arc-consistency of a network, as we see in the example below.

Example 3.2.4 Consider, now, the matching diagram of the three-variable constraint network depicted in Figure 3.3a. Without knowing the nature of the constraint between y and z, we can see that the two are arc-consistent relative to one another because each value in the domains of the two variables can be matched to an element from the other. However, their arc-consistency is violated in the process of making the adjacent constraints arc-consistent. Specifically, to make $\{x, z\}$ arc-consistent, we must delete a value from the domain of x, which will leave x no longer arc-consistent relative to y. Consequently, Revise may need to be applied more than once to each constraint until there is no change in the domain of any variable in the network. □

Algorithm *Arc-consistency-1 (AC-1)*, a brute-force algorithm that enforces arc-consistency over the network, is given in Figure 3.4. The algorithm applies the Revise rule to all pairs of variables that participate in a constraint until a full cycle ensures that no domain has been altered. The arc-consistent equivalent of the network in the matching diagram Figure 3.3a is given in Figure 3.3b.

Occasionally, arc-consistency enforcing algorithms may discover inconsistency.

Example 3.2.5 Consider a binary network over three variables $\{x, y, z\}$, where the domains of all the variables are $\{1, 2, 3\}$ and the constraints are $x < y$, $y < z$, $z < x$. Once we have finished processing a pair of variables, we can say both that the variables are arc-consistent relative to one another and that the constraint itself is arc-consistent. Applying arc-consistency to the variables that participate in constraints in the sequence R_{xy}, R_{yz}, R_{zx}, we get first (when revising R_{xy} in both directions) that D_x is reduced to $\{1, 2\}$ and D_y to $\{2, 3\}$. Then, processing constraint R_{yz}, the domain of y is further reduced to $\{2\}$ and the domain of z to $\{3\}$. When R_{zx} is processed, the domain of z becomes empty. Subsequent processing will empty the domains of y and z, and we conclude that the network is inconsistent.

□

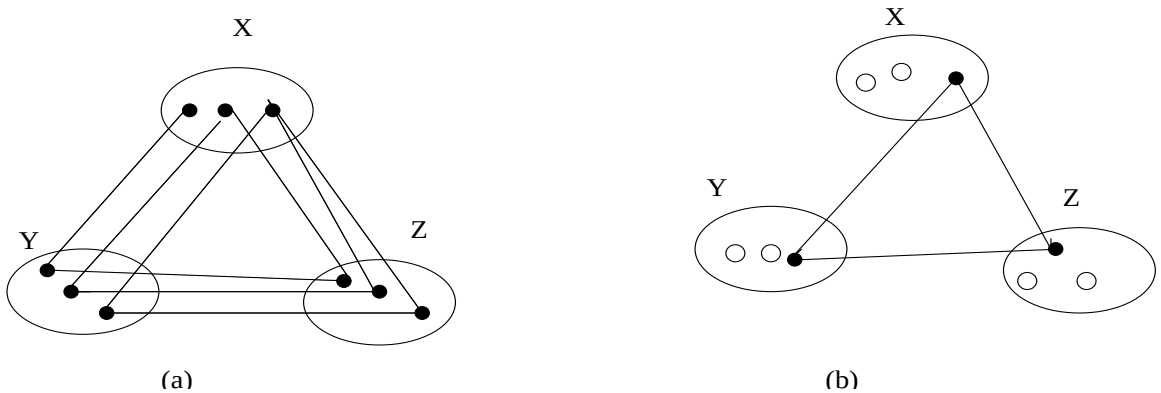

Figure 3.3: (a) Matching diagram describing a network of constraints that is not arc-consistent (b) An arc-consistent equivalent network.

AC-1($\mathcal{R}$)
input: a network of constraints $\mathcal{R} = (X, D, C)$
output: $\mathcal{R}'$ which is the loosest arc-consistent network equivalent to $\mathcal{R}$
1. **repeat**
2. **for** every pair $\{x_i, x_j\}$ that participates in a constraint
3. Revise$((x_i), x_j)$ (or $D_i \leftarrow D_i \cap \pi_i(R_{ij} \bowtie D_j)$)
4. Revise$((x_j), x_i)$ (or $D_j \leftarrow D_j \cap \pi_j(R_{ij} \bowtie D_i)$)
5. **endfor**
6. **until** no domain is changed

Figure 3.4: Arc-consistency-1 (AC-1)

As we have seen, algorithm *AC-1* generates an equivalent arc-consistent network, and when an empty domain is encountered, we concluded that the network has no solution. AC-1 has the following complexity:

Proposition 3.2.6 *Given a constraint network $\mathcal{R}$ having n variables, with domain sizes bounded by k, and e binary constraints, the complexity of AC-1 is $O(enk^3)$.*

Proof: In AC-1, one cycle through all the binary constraints (steps 2-5) takes $O(ek^2)$. Since, in the worst case, one cycle may cause the deletion of just one value from one domain, and since, overall, there are nk values, the maximum number of such cycles is nk, resulting in the overall bound of $O(n \cdot ek^3)$. □

AC-3($\mathcal{R}$)
input: a network of constraints $\mathcal{R} = (X, D, C)$
output: $\mathcal{R}'$ which is the largest arc-consistent network equivalent to $\mathcal{R}$
1. **for** every pair $\{x_i, x_j\}$ that participates in a constraint $R_{ij} \in \mathcal{R}$
2. $\quad$ *queue* $\leftarrow$ *queue* $\cup \{(x_i, x_j), (x_j, x_i)\}$
3. **endfor**
4. **while** *queue* $\neq \{\}$
5. $\quad$ select and delete (x_i, x_j) from *queue*
6. $\quad$ *Revise*$((x_i), x_j)$
7. $\quad$ **if** *Revise*$((x_i), x_j)$ causes a change in D_i
8. $\quad\quad$ **then** *queue* $\leftarrow$ *queue* $\cup \{(x_k, x_i), k \neq i,\ k \neq j\}$
9. $\quad$ **endif**
10. **endwhile**

Figure 3.5: Arc-consistency-3 (AC-3)

Algorithm AC-1 can be improved. There is no need to process all the constraints if only a few domains were reduced in the previous round. The improved version of arc-consistency establishes and maintains a queue of constraints to be processed. Initially, each pair of variables that participates in a constraint is placed in the queue twice (once for each ordering of the pair of variables). Once an ordered pair of variables is processed, it is removed from the queue, and is placed back in the queue only if the domain of its second variable is modified as a result of the processing of adjacent constraints. We name this algorithm *arc-consistency-3 (AC-3)* to conform to the usage already established in the community. A previous improvement called *AC-2* provides only a minor advancement step in between these two versions. Algorithm *(AC-3)*, which uses a queue data structure, is presented in Figure 3.5.

Example 3.2.7 Consider a three-variable network: z, x, y with $D_z = \{2, 5\}$, $D_x = \{2, 5\}$, $D_y = \{2, 4\}$. There are two constraints: R_{zx}, specifying that z evenly divides x, and R_{zy}, specifying that z evenly divides y. The constraint graph of this problem is depicted in Figure 3.6a. Assume that we apply AC-3 to the network. We put $(z, x), (x, z), (z, y)$, and (y, z) onto the queue. Processing the pairs (z, x) and (x, z) does not change the problem, since the domains of z and x are already arc-consistent relative to R_{zx}. When we process (z, y), we delete 5 from D_z, and consequently place (x, z) back on the queue. Processing (y, z) causes no further change, but when (x, z) is revised, 5 will be deleted from D_x. At this point, no further constraints will be inserted into the queue (remember, no constraint

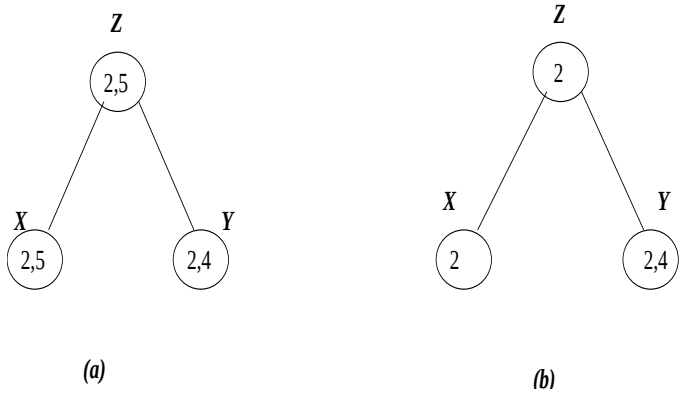

Figure 3.6: A three variable network, with two constraints: z divides x and z divides y (a)before and (b)after AC-2 is applied.

already in the queue needs to be inserted), and the algorithm terminates with the domains $D_x = \{2\}$, $D_z = \{2\}$, and $D_y = \{2, 4\}$. □

Algorithm AC-3 processes each constraint at most $2k$ times, where k bounds the domain size, since each time it is re-introduced into the queue, the domain of a variable in its scope has just been reduced by at least one value, and there are at most $2k$ such values. Since there are e binary constraints, and processing each one takes $O(k^2)$, we can conclude that:

Proposition 3.2.8 *The time complexity of* AC-3 *is* $O(ek^3)$. □

Is AC-3's performance optimal? It seems that no algorithm can have time complexity below ek^2, since the worst case of merely verifying the arc-consistency of a network requires ek^2 operations[2]. Indeed, algorithm Arc-consistency-4 (AC-4) achieves this optimal performance. AC-4 does not use Revise or the composition operator as an atomic operation. Instead, it exploits the underlying micro structure of the constraint relation and tunes its operation to that level.

AC-4 associates each value a_i in the domain of x_i with the amount of *support from variable* x_j, that is, the number of values in the domain of x_j that are consistent with value a_i. A value a_i is then removed from the domain D_i if it has no support from some neighboring variable. The algorithm maintains a *LIST* of currently unsupported variable-value pairs, a counter array $counter(x_i, a_i, x_j)$ of supports for a_i from x_j, and an array $S_{(x_j,a_j)}$ that points to all values in other variables supported by $< x_j, a_j >$. In each step, the algorithm picks up an unsupported value from the LIST, adds it to the *removed* list

[2]For a proof for the special case of tree-networks see [77].

AC-4($\mathcal{R}$)
input: a network of constraints $\mathcal{R}$
output: An arc-consistent network equivalent to $\mathcal{R}$
1. Initialization: $M \leftarrow \emptyset$
2. initialize $S_{(x_i,c_i)}$, $counter(i, a_i, j)$ for all R_{ij}
3. **for** all counters
4. **if** $counter(x_i, a_i, x_j) = 0$ (if $< x_i, a_i >$ is unsupported by x_j)
5. **then** add $< x_i, a_i >$ to $LIST$
6. **endif**
7. **endfor**
8. **while** $LIST$ is not empty
9. choose $< x_i, a_i >$ from LIST, remove it, and add it to M
10. **for** each $< x_j, a_j >$ in $S_{(x_i,a_i)}$
11. decrement $counter(x_j, a_j, x_i)$
12. **if** $counter(x_j, a_j, x_i) = 0$
13. **then** add $< x_j, a_j >$ to $LIST$
14. **endif**
15. **endfor**
16. **endwhile**

Figure 3.7: Arc-consistency-4 (AC-4)

M, and updates all the supports of potentially affected values. Those values that became unsupported as a result are placed in the $LIST$. If a_j is unsupported, the counters of values that it supports will be reduced. Algorithm AC-4 is given in Figure 3.2.

$'$ where all variables domains are reduced by values in M.

Arc-consistency-4 (AC-4)

Example 3.2.9 Consider the problem in Figure 3.6. Initializing the $S_{(x,a)}$ arrays (indicating all the variable-value pairs that each $< x, a >$ supports), we have:
$S_{(z,2)} = \{< x, 2 >, < y, 2 >, < y, 4 >\}$, $S_{(z,5)} = \{< x, 5 >\}$, $S_{(x,2)} = \{< z, 2 >\}$, $S_{(x,5)} = \{< z, 5 >\}$, $S_{(y,2)} = \{< z, 2 >\}$, $S_{(y,4)} = \{< z, 2 >\}$.
For counters we have: $counter(x, 2, z) = 1$, $counter(x, 5, z) = 1$, $counter(z, 2, x) = 1$, $counter(z, 5, x) = 1$, $counter(z, 2, y) = 2$, $counter(z, 5, y) = 0$, $counter(y, 2, z) = 1$, $counter(y, 4, z) = 1$. (Note that we do not need to add counters between variables that

are not directly constrained, such as x and y.) Finally, $List = \{< z, 5 >\}$, $M = \emptyset$. Once $< z, 5 >$ is removed from $List$ and placed in M, the counter of $< x, 5 >$ is updated to $counter(x, 5, z) = 0$, and $< x, 5 >$ is placed in $List$. Then, $< x, 5 >$ is removed from $List$ and placed in M. Since the only value it supports is $< z, 5 >$ and since $< z, 5 >$ is already in M, the $List$ remains empty and the process stops. □

The initialization step that creates the counter of supports and the pointers requires, at most, $O(ek^2)$ steps. The number of elements in $S_{(xj,a_j)}$ is on the order of ek^2. We conclude, therefore:

Proposition 3.2.10 *The time and space complexity of AC-4 is* $O(ek^2)$.

Proof: (exercise 4).

Since worst-case complexity often overestimates, and since average-case analysis is hard to achieve, it is sometime useful to introduce, more refined parameters into the analysis. Instead of using $O(k^2)$ – which is the size of the universal relation (remember that arc-consistency is relevant only to the binary constraints) – as a bound for each relation size, we can use a *tightness* parameter t that stands for the maximum number of tuples in each binary relation. Frequently the constraints can be quite tight, as in the case of functional constraints where $t = O(k)$.

Revisiting our worst-case analysis, the Revise procedure can be modified to have a complexity of $O(t)$. Consequently, the complexity of AC-1 is modified to $O(n \cdot k \cdot e \cdot t)$, that of AC-3 to $O(e \cdot k \cdot t)$, and that of AC-4 to $O(e \cdot t)$.

Analyzing the best-case performance of these algorithms may also provide insight. The best case of AC-1 and AC-3 is $e \cdot k$ steps, because the problem may already be arc-consistent. The best case of AC-4 remains at ek^2, which is the time necessary to create the special data structures in its initialization phase. Consequently, when the constraints are loose (i.e., when t is closest to k^2), AC-1 and AC-3 may frequently outperform AC-4, even though AC-4 is optimal in the worst case.

3.3 Path-consistency

As we saw in Example 3.2.5, arc-consistency can sometimes decide inconsistency by discovering an empty domain. However, arc-consistency is not complete for deciding consistency because it makes inferences based on a single (binary) constraint and single domain constraints.

Example 3.3.1 Consider the example we mentioned at the outset having three variables x, y, z with respective domains $\{red, blue\}$ and constrained by $x \neq y$, $y \neq z$, $z \neq x$.

This constraint network is arc-consistent without reducing any domains, and therefore AC enforcement will not reveal the network's inconsistency. Although the constraint R_{xy},which is $x \neq y$, allows the assignment $(< x, red >, < y, blue >)$, no color for z will be consistent with both $< x, red >$ and $< y, blue >$. In this case, we say that R_{xy} is not *path-consistent* relative to z, and, more generally, that the subnetwork associated with the three variables $\{x, y, z\}$ is not 3-consistent. Indeed, we can infer something about this network: since our domain size is 2, $x \neq y$ and $y \neq z$ imply $x = z$, but $x = z$ is not explicit in this network. Not only that, it contradicts another explicit constraint. □

The consistency level violated here is called *path-consistency*, requiring consistency relative to paths of length 3 (the path from x to y that goes through z). It is interesting to note that in the minimal network path-consistency is maintained; any consistent pair of values can be consistently extended by any third variable (see exercise 6).

In general, the notion of path-consistency involves inferences using subnetworks having three variables. However, since the original definition of path-consistency involves only binary constraints, it is relevant only to the binary constraints of the network. An extension of the definition to non-binary constraints is presented later; it constitutes only a minor change which is relevant when the network also has ternary constraints.

Definition 3.3.2 (Path-consistency) *Given a constraint network $\mathcal{R} = (X, D, C)$, a two variable set $\{x_i, x_j\}$ is path-consistent relative to variable x_k if and only if for every consistent assignment $(< x_i, a_i >, < x_j, a_j >)$ there is a value $a_k \in D_k$ s.t. the assignment $(< x_i, a_i >, < x_k, a_k >)$ is consistent and $(< x_k, a_k >, < x_j, a_j >)$ is consistent. Alternatively, a binary constraint R_{ij} is path-consistent relative to x_k iff for every pair $(a_i, a_j), \in R_{ij}$, where a_i and a_j are from their respective domains, there is a value $a_k \in D_k$ s.t. $(a_i, a_k) \in R_{ik}$ and $(a_k, a_j) \in R_{kj}$. A subnetwork over three variables $\{x_i, x_j, x_k\}$ is path-consistent iff for any permutation of (i, j, k), R_{ij} is path consistent relative to x_k. A network is path-consistent iff for every R_{ij} (including universal binary relations) and for every $k \neq i, j$ R_{ij} is path-consistent relative to x_k.*

Pictorially, a connected pair of points in the matching diagram (denoting a legal pair of values in R_{xy}) satisfies path-consistency iff, the connected pair can be extended to a triangle with a third value as is shown in Figure 3.8b. When a three-variable subnetwork is not path-consistent, we can enforce path-consistency by making the necessary inferences. So, if in the input specification there is no constraint between x and y (meaning that everything is allowed), we can deduce a new constraint by the length-3 path from x to y through z. In this case, the necessary inferences are adding binary constraints or tightening binary constraints (deleting tuples from the relation). In the coloring example (see the matching diagram in Figure 3.8a), if we attempt to make R_{xy} path-consistent relative to z, this implies emptying the constraint R_{xy} and declaring inconsistency.

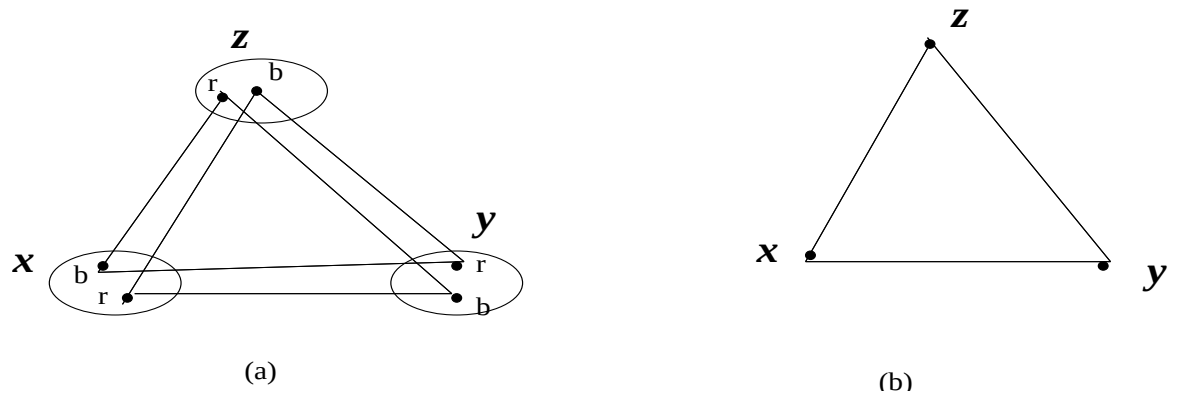

Figure 3.8: (a) The matching diagram of a 2-value graph coloring problem. (b) Graphical picture of path-consistency using the matching diagram.

Alternatively, consider again the network of three variables $\{x, y, z\}$ where all variables have the same domain and the equality constraints: $R_{xz}: \ x = z$, and $R_{yz}: \ y = z$. To make this network path-consistent, we should infer and add to the network the constraint $R_{xy}: \ x = y$.

To use an analog of Revise for arc-consistency, we define a procedure *Revise-3*$((x, y), z)$ (Figure 3.9). This procedure takes a pair of variables (x, y) and their constraint which we wish to modify, R_{xy} (and which can also be the universal constraint, allowing all possible pairs), and a third variable, z, and returns the loosest constraint R'_{xy} that satisfies path-consistency. Revise-3 tests if each pair of consistent values in R_{xy} can be extended consistently to a value of z, and if not, it deletes the violating pair.

REVISE-3$((x, y), z)$
input: a three-variable subnetwork over (x, y, z), R_{xy}, R_{yz}, R_{xz}.
output: revised R_{xy} path-consistent with z.
1. **for** each pair $(a, b) \in R_{xy}$
2. **if** no value $c \in D_z$ exists such that $(a, c) \in R_{xz}$ and $(b, c) \in R_{yz}$
3. **then** delete (a, b) from R_{xy}.
4. **endif**
5. **endfor**

Figure 3.9: Revise-3

.

Revise-3$((x, y), z)$ can be expressed succinctly as the composition,

$$R_{xy} \leftarrow R_{xy} \cap \pi_{xy}(R_{xz} \bowtie D_z \bowtie R_{zy})$$

As you can see, Revise-3 makes the minimal necessary changes to make R_{xy} path-consistent relative to z. The reader can verify that its performance is characterized by:

Proposition 3.3.3 *The complexity of* Revise-3 *is* $O(k^3)$ *and* $O(t \cdot k)$ *where* k *bounds the domain size and* t *bounds the constraint tightness.* □

The claim of the proposition assumes that testing the consistency $(a, b) \in R_{xy}$ can be done in constant time using hash tables. Otherwise a *log* k factor should be introduced. When using the tightness parameter t, the worst-case complexity of Revise-3 is $O(t \cdot k)$ and its best-case performance may take just t steps.

Path-consistency can be applied to a subnetwork, or it can be fully enforced with respect to all subnetworks of size 3, yielding a completely *path-consistent network*. The following two algorithms, *Path-consistency-1* (PC-1) and *Path-consistency-3* (PC-3) enforce path-consistency on a network in an manner analogous to AC-1 and AC-3, respectively. PC-1 applies Revise-3 in a brute-force manner to every triplet of variables until there is one full cycle with no change, as shown in Figure 3.10. Its complexity analysis is similar to AC-1's.

PC-1($\mathcal{R}$)
input: a network $\mathcal{R} = (X, D, C)$.
output: a path consistent network equivalent to $\mathcal{R}$.
1. **repeat**
2. **for** $k \leftarrow 1$ to n
3. **for** $i, j \leftarrow 1$ to n
4. $R_{ij} \leftarrow R_{ij} \cap \pi_{ij}(R_{ik} \bowtie D_k \bowtie R_{kj})$/* $(Revise - 3((i, j), k))$
5. **endfor**
6. **endfor**
7. **until** no constraint is changed.

Figure 3.10: Path-consistency-1 (PC-1)

Proposition 3.3.4 *Given a network* $\mathcal{R}$*, when* k *bounds the domain sizes and* t *bounds the tightness of the constraints, PC-1 generates an equivalent path-consistent network, with complexity* $O(n^5k^5)$ *or* $O(n^5 \cdot t^2 \cdot k)$.

Proof: Since in each cycle (steps 3, 4 and 5) we may eliminate only one pair of values from one constraint, the number of cycles is $O(n^2k^2)$ (resp. $O(n^2 \cdot t)$). Since we have $O(n^3)$ triplets of variables, and since processing each triplet by Revise-3 is $O(k^3)$ (resp. $O(t \cdot k)$), each cycle costs $O(n^3k^3)$ (resp. $O(n^3 \cdot t \cdot k)$), yielding the above bound. □

Algorithm PC-3 (Figure 3.11) improves on PC-1 by maintaining a queue of ordered triplets to be processed (or reprocessed). Once a constraint R_{ij} is modified, thus deleting a pair of its values, all the triplets involving x_i or x_j, and any third variable x_k, are reprocessed.

Note that in order to make the network fully path-consistent, all ordered triplets should be included in the initial queue, even those containing pairs of variables that have no direct binary constraint.

PC-3($\mathcal{R}$)
input: a network $\mathcal{R} = (X, D, C)$.
output: $\mathcal{R}'$ a path consistent network equivalent to $\mathcal{R}$.
1. $Q \leftarrow \{(i,k,j) \mid 1 \leq i < j \leq n, 1 \leq k \leq n, k \neq i, k \neq j\ \}$
2. **while** Q is not empty
3. select and delete a 3-tuple (i,k,j) from Q
4. $R_{ij} \leftarrow R_{ij} \cap \pi_{ij}(R_{ik} \bowtie D_k \bowtie R_{kj})$ /* (Revise-3$((i,j),k)$)
5. **if** R_{ij} changed then
6. $Q \leftarrow Q \cup \{(l,i,j)(l,j,i) \mid 1 \leq l \leq n, l \neq i, l \neq j\}$
7. **endwhile**

Figure 3.11: Path-consistency-3 (PC-3)

Proposition 3.3.5 *Given a network $\mathcal{R}$, algorithm PC-3 generates an equivalent path-consistent network, and its complexity is $O(n^3k^5)$ ($O(n^3 \cdot t^2 \cdot k)$, resp.), where k and t represent the tightness of domains and constraints, respectively.*

Proof: exercise 12. □

Example 3.3.6 Consider the graph-coloring problem on a four-variable network where the domains contain only two possible colors $D_1 = D_2 = D_3 = D_4 = \{red, blue\}$, and the constraints are $x_1 \neq x_2$, $x_2 \neq x_3$, $x_3 \neq x_4$, $x_4 \neq x_1$. The constraint graph of this problem is depicted in Figure 3.12a. The network is currently arc-consistent, but not path-consistent. For example; the universal constraint between x_3 and x_1 allows the assignment $x_1 = red$ and $x_3 = blue$, yet there is no assignment to x_4 satisfying the unequal constraints with this instantiation. Enforcing path-consistency requires adding the constraint $x_1 = x_3$. Path-consistency will similarly add the constraint $x_2 = x_4$. The path-consistent constraint

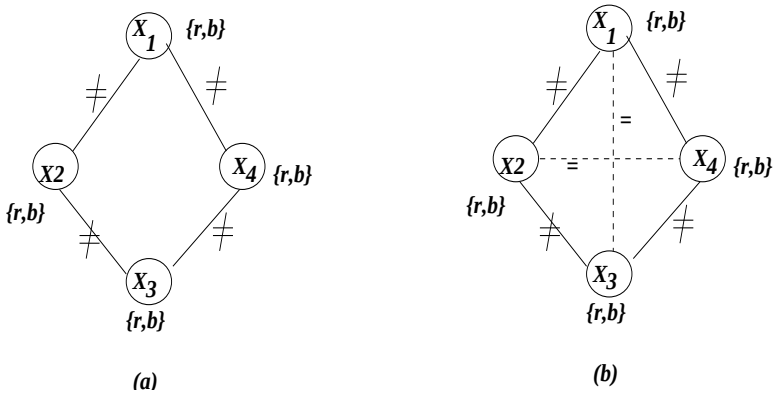

Figure 3.12: A graph-coloring graph (a) before path-consistency (b) after path-consistency

network version of this example is depicted in Figure 3.12b. If we generate a path-consistent network by applying PC-1 to the original network, the algorithms' first cycle applies Revise-3 to four triplets, generating the two equality constraints. A full cycle will then be executed to verify that nothing changes. This verification requires a second processing of each triplet. On the other hand, if we enforce path-consistency by PC-3, we may be able to process each triplet only once, assuming the right ordering is picked. If we apply Revise-3 first to (x_1, x_3, x_2), that is, to the universal constraint between x_1 and x_3, and then to (x_2, x_4, x_1) each triplet would be processed just once. □

Like its arc-consistent counterpart, PC-3 is not optimal, though we can devise an optimal algorithm, akin to AC-4. It would require operating on the relation level and maintaining *supports* for pairs of values. An algorithm exploiting such low-level consistency maintenance, which we will call *PC-4*, is available [211], and its complexity bound is $O(n^3k^3)$ or $O(n^3tk)$. It is an optimal algorithm, since even verifying path-consistency has that lower bound, namely it is $\Omega(n^3k^3)$.

Regarding best-case performance, we observe that PC-1, PC-3 and PC-4 have properties that parallel those of arc-consistency. Algorithms PC-1 and PC-3 can be as good as $n^3 \cdot t$ and $n^3 \cdot k^2$ respectively, while algorithm PC-4 (which was not presented explicitly) requires an order of n^3k^3 (or $n^3 \cdot t \cdot k$) even in the best case because of its initialization (see exercises).

Let's conclude our introduction to path-consistency by giving an alternative definition that may explain the origin of the term.

Definition 3.3.7 (A path-consistent constraint) *A constraint R_{ij} is path-consistent, relative to the path of length m through the nodes $(i = i_0, i_1, ..., i_m = j)$, if for any pair $(a_i, a_j) \in R_{ij}$ there is a sequence of values $a_{il} \in D_{i_l}$ such that $(a_i = a_{i_0}, a_{i_1}) \in R_{i_0 i_1}$, $(a_{i_0}, a_{i_1}) \in R_{i_0 i_1}$, ..., and $(a_{i_{m-1}}, a_{i_m} = a_j) \in R_{i_{m-1} i_m}$.*

For a complete constraint graph the two definitions are the same. However, if we require path-consistency to hold relative to triangles in the constraint graph only, then the two above-referenced definitions are not the same (see exercise 14).

3.4 Higher levels of i-consistency

Arc- and path-consistency algorithms process subnetworks of size 2 and 3 respectively. We're now ready to generalize the concept of local consistency to subnetworks of size i. In this case non-binary constraints also come into play.

Definition 3.4.1 (i-consistency, global consistency) *Given a general network of constraints $\mathcal{R} = (X, D, C)$, a relation $R_S \in C$ where $|S| = i - 1$ is i-consistent relative to a variable y not in S iff for every $t \in R_S$, there exists a value $a \in D_y$, s.t. (t, a) is consistent. A network is* i-consistent *iff given any consistent instantiation of any $i - 1$ distinct variables, there exists an instantiation of any ith variable such that the i values taken together satisfy all of the constraints among the i variables. A network is* strongly i-consistent *iff it is j-consistent for all $j \leq i$. A strongly n-consistent network, where n is the number of variables in the network, is called* globally consistent.

Globally consistent networks are characterized by the property that any consistent instantiation of a subset of the variables can be extended to a consistent instantiation of all of the variables without encountering any dead-ends.

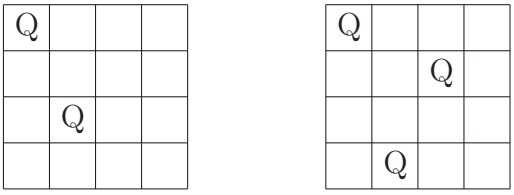

Figure 3.13: (a) Not 3-consistent; (b) Not 4-consistent

Example 3.4.2 Consider the constraint network for the 4-queens problem. We see that the network is 2-consistent since, given that we have placed a single queen on the board, we can always place a second queen on any remaining column such that the two queens do not attack each other. This network is not 3-consistent; however, given the consistent placement of two queens shown in Figure 3.13a, there is no way to place a queen in the third column that is consistent with the previously placed queens. Similarly, the network is not 4-consistent (Figure 3.13b). □

Enforcing higher levels of consistency requires recording constraints on more than two variables. Consider first the i-level Revise rule. If a consistent instantiation of $i-1$ variables cannot be consistently extended to an i^{th} variable, we need to *rule out* that instantiation, labelling it as a *nogood*. This means that we should record an explicit constraint over the scope of these $i-1$ variables that excludes that particular nogood assignment. As a consequence of this exclusion, even if the input network is binary, enforcing i-consistency may result in a non-binary network having constraints of arity $i-1$. For example, to make the 4-queen example 4-consistent we need to add, among other constraints, the ternary constraint over x_1, x_2, x_3 that forbids the tuple $(< x_1, 1 >< x_2, 4 >< x_3, 2 >)$. The *Revise-i* operation carries out this task, which is defined in Figure 3.14.

REVISE-$i(\{x_1, x_2, ..., x_{i-1}\}, x_i)$
input: a network $\mathcal{R} = (X, D, C)$
output: a constraint R_S ($S = \{x_1, ..., x_{i-1}\}$) which is i-consistent relative to x_i.
1. **for** each instantiation $\bar{a}_{i-1} = (< x_1, a_1 >, < x_2, a_2 >, ..., < x_{i-1}, a_{i-1} >)$ do,
2. **if** no value of $a_i \in D_i$ exists s.t. $(\bar{a}_{i-1}, a_i)$ is consistent
   then delete $\bar{a}_{i-1}$ from R_S
   (Alternatively, let $\mathcal{S}$ be the set of all subsets of $\{x_1, ..., x_i\}$ that contain x_i and appear as scopes of constraints of $\mathcal{R}$, then
   $R_S \leftarrow R_S \cap \pi_S(\bowtie_{S' \subseteq \mathcal{S}} R_{S'})$)
3. **endfor**

Figure 3.14: Revise-i

The complexity of brute-force Revise-i is $O(k^i)$, if we assume that the input specification includes binary constraints only. However, when arbitrary constraints are involved, Revise-i by itself can be as complex as $O((2k)^i)$ because there may be $O(2^i)$ constraints that need to be tested for each tuple.

Several algorithms for enforcing i-consistency have been developed. When Revise-i is incorporated into a brute-force algorithm enforcing i-consistency over a given network $\mathcal{R}$, all subsets of size $i-1$ need to be modified, requiring $O(n^i(2k)^i) = O((2nk)^i)$ steps. These steps should be repeated until a complete cycle produces no change. The number of such cycles equals $O(n^i k^{i-1})$, yielding overall time complexity of $O((nk)^{2i}2^i)$ and space complexity of $O(n^i k^i)$. On the other hand, a lower bound for enforcing i-consistency is $\Omega((nk)^i)$ (for more details see [65]). Indeed there exists an optimal algorithm that achieves this bound (using complex data structures) that is analogous to AC-4 and PC-4 [51]. In summary,

I-CONSISTENCY($\mathcal{R}$)
input: a network $\mathcal{R}$.
output: an i-consistent network equivalent to $\mathcal{R}$.
1. **repeat**
2. **for** every subset $S \subseteq X$ of size $i-1$, and for every x_i, do
3. let $\mathcal{S}$ be the set of all subsets of $\{x_1, ..., x_i\}$ in $scheme(\mathcal{R})$ that contain x_i
4. $R_S \leftarrow R_S \cap \pi_S(\bowtie_{S' \in \mathcal{S}} R_{S'})$ (this is Revise-i(S, x_i))
6. **endfor**
7. **until** no constraint is changed.

Figure 3.15: i-consistency-1

Theorem 3.4.3 (complexity of i-consistency) *The time and space complexity of brute-force i-consistency $O(2^i(nk)^{2i})$ and $O(n^ik^i)$, respectively. A lower bound for enforcing i-consistency is $\Omega(n^ik^i)$.* □

Many additional variants of local-consistency algorithms exist in the literature. We will revisit this topic when we introduce *relational consistency* in Chapter 8. There we characterize consistency level by the number of constraints involved rather than by the number of variables. We conclude with a simple extension of arc-consistency to non-binary constraints. This extension is a special case of relational consistency elaborated in Chapter 8.

3.4.1 3-consistency and Path-consistency

For binary constraint networks, 3-consistency is identical to path-consistency. When the network also has ternary constraints, the definition of 3-consistency requires also testing ternary constraints.

Example 3.4.4 Suppose a constraint network involves three variables x, y, z having domains $\{0, 1\}$ and a single ternary constraint $R_{xyz} = \{(0, 0, 0)\}$. Application of the path-consistency algorithm will produce nothing since there are no binary constraints to test; the network is already path-consistent. However, the network is *not* 3-consistent. While we can assign the values $(\langle x, 1\rangle, \langle y, 1\rangle)$ (since there is no constraint), we cannot extend this assignment to z in a way that satisfies the given ternary constraint. Indeed, if we apply 3-consistency to this network we will add at least the constraint $R_{xy} = \{(< x, 0 >< y, 0 >)\}$ in addition to the constraint $R_x = \{(< x, 0 >)\}$. □

You can see here that the Revise-3 operation, as depicted in Figure 3.14, will indeed test consistency relative to ternary constraints as well as binary ones.

3.5 Arc-consistency for non-binary constraints

3.5.1 Generalized arc-consistency

So far, the notion of constraint propagation, as reflected by arc-consistency, is restricted to pairs of variables only. In applications, constraints are most often non-binary, so an extension of this simplest form of constraint propagation to the non-binary case is called for. There are many simple local inferences that can be made between a single non-binary constraint and a single variable. For example, a constraint such as $x+y+z \leq 15$, together with a domain constraint $z \geq 13$, can immediately lead one to the restrict y to $y \leq 2$ and x to $x \leq 2$. As another example, we can have a non-binary party constraint saying that if Alex (A) and Bill (B) go to the party, then Gill goes to the party (G), that is, $A \wedge B \rightarrow G$. If we learn that Gill did not go to the party (that is, $\neg G$) we can conclude $\neg A \vee \neg B$. In both cases inference is local, relying on a single constraint and a single variable. In one case, local propagation reduces only the domain (often called domain-reduction). In the others, we can infer constraints on more than a single variable.

Hence an extension of arc-consistency to non-binary constraints can be done in two complementary ways. Both definitions take a constraint and a single variable. The first extension is called *generalized arc-consistency* and leads to concepts such as *domain reduction* or *domain propagation*.

Definition 3.5.1 (generalized arc-consistency) *Given a constraint network $\mathcal{R} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, with $R_S \in C$, a variable x is* arc-consistent *relative to R_S if and only if for every value $a \in D_x$ there exists a tuple $t \in R_S$ such that $t[x] = a$. t can be called a* support *for a. The constraint R_S is called arc-consistent iff it is arc-consistent relative to each of the variables in its scope. A constraint network is arc-consistent if all its constraints are arc-consistent.*

We can enforce generalized arc-consistency on a constraint by shrinking the domains of its variables. A simple extension of the Revise procedure can be obtained using the same equation as in 3.1, applied to a non-binary relation, namely by:

$$D_x \leftarrow D_x \cap \pi_x(R_S \bowtie D_x). \tag{3.2}$$

Since each value in D_x is compared, in the worst case, with each tuple of R_S we get:

Proposition 3.5.2 *The complexity of the rule in Eq. (3.2) is $O(t \cdot k)$ where k bounds the domain size and t is the constraint tightness. Note that $t \leq k^r$ when r is the constraint scope size.* □

The second extension of arc-consistency is complementary in the sense that it records the inferred constraint on the entire scope of the constraint *but* except for the single variable involved in the inference. This inference is called *relational arc-consistency*, and it can be enforced for every constraint R_S and a variable $x \in S$ by:

$$R_{S-\{x\}} \leftarrow \pi_{S-\{x\}}(R_S \bowtie D_x). \tag{3.3}$$

3.5.2 Global constraints

So far we have discussed constraint propagation relative to specified input constraints that are assumed to be available in relational form. However the task of modeling real applications as constraint problems requires developing specialized propagation algorithms for frequently used constraints, either because a relational description is unrealistic, or because standard propagation rules are inefficient, not effective or both.

The most well known example is the *alldifferent* constraint: *alldfferent*$(x_1, x_2, ...x_l)$ which requires that the values assigned to each variable among the set will be mutually different. This constraint appears in almost every assignment and resource allocation problem and it can be expressed by a collection of binary not-equal constraints. Applying arc-consistency over such a binary network yields little or no affect because such networks are often already arc-consistent (unless there are single-valued domains). Consequently, specialized filtering algorithms were developed.

Many such constraints that appear often in modeling, and for which specialized domain reduction propagation algorithms have been developed are called *global constraints*. Typically, a global constraint is defined over any size scope (e.g., alldifferent). The level of consistency enforced on such constraints can be described as (generalized) arc-consistency relative to their underlying implicit relation. Since it takes advantage of the structure of the global constraint, the method for achieving the targeted level of consistency is specialized and domain-dependent.

Consider the problem of scheduling talks of speakers in a conference. Each talk is a one-hour slot and the speakers are available only at certain time slots. Assume we have the following speakers: Alfred (A) who can talk in slots $\{3, 4, 5, 6\}$, Bob (B) in $\{3, 4\}$, Cindy (C) in $\{2, 3, 4, 5\}$, Debby (D) in $\{2, 3, 4\}$, Eldridge (E) in $\{3, 4\}$ and Fred (F) in $\{1, 2, 3, 4, 5, 6\}$. There can be many additional constraints, such as that Cindy and Debby should teach one following the other, etc. Modelling the problem can be done by a set of binary *not-equal* constraints between every pair of distinct variables: $A \neq B$, $B \neq C$, etc, for every pair of variables. A quick look reveals that all these constraints are

arc-consistent, and therefore no domain pruning will occur by enforcing arc-consistency. Suppose now that we view the set of not-equal binary constraints as a single *global* alldifferent constraint $R(A, B, C, D, E, F)$ on all the variables. Namely, it denotes the implicit constraint, representing all solutions to the unary (domain constraints) and binary (not-equal) constraints.

If we apply generalized arc-consistency to this *global* constraint we will see that the domain of the variables will be reduced considerably. For example, $D_A = \{6\}$, $D_F = \{1\}$, etc. (see exercises). Clearly, a brute-force algorithm for achieving this level of consistency amounts to solving a subproblem and projecting its set of solutions on each single domain using rule (3.2). This process is quite costly however, and in general it makes little practical sense. However, some specialized constraints may allow efficient processing for achieving that level of consistency.

In particular, it can be shown that generalized arc-consistency over the global alldifferent constraint can be accomplished by a bi-partite matching algorithm between variables and values leading an algorithm whose complexity is $O(k \cdot n^{1.5})$ Additional specialized global constraint (e.g., global cardinality constraints) were also shown to posses efficient arc-consistency methods.

The list of global constraints is long. Among the most popular are: 1. the alldifferent constraints discussed above, 2. a constraint of sum making one variable equal to the sum of k other variables, 3. Global cardinality constraint that generalize the alldifferent constraint. 4. The *commulative constraint enforces at each point in time, the commulative resource consumption of a set of tasks that overlaps, does not exceed the specified capacity. Each task has a start, a duration and a resource consumption which are all domain variables. 5. The* cycle *constraint cycle*$(N, x_1, ..., x_m)$ *is a permutation, namely all the variables are pairwise distinct and each takes a value in* $\{1, 2, ..., m\}$ *which contains* N *cycles.*

In summary, a global constraint $C = \{C(i)\}$ *is a family of scope-parameterized constraints, (normally* $i \geq 2$*), where* $C(i)$ *is a constraint whose relation is often defined implicitly by either a natural language statement, or as a set of solutions to a subproblem defined by lower arity explicit constraints (e.g., alldifferent). It is associated with one or more specialized propagation algorithms trying to achieve generalized arc-consistency relative to* $C(i)$ *(or an approximation of it) in a way that is more efficient than a brute-force approach.*

3.5.3 Bounds consistency

In some cases the application of arc-consistency is too costly. This is the case when the domains of the variables are large sets of integers and occurs often (but not only) when developing arc-consistency algorithms for the kinds of global constraints discussed above.

In such cases a weaker notion of consistency called bounds-consistency *appears to be highly cost-effective. It is less costly than applying full arc-consistency on either input or global constraints. The idea is to bound the domain of each variable by an interval and make sure that only the end-points of the intervals obey the arc-consistency requirement. If not, the upper and lower bounds of the intervals can be tightened until bounds consistency is achieved.*

Definition 3.5.3 (bounds consistency) *Given a constraint C over a scope S and domain constraints, a variable $x \in S$ (when S is a well defined ordered set), is* bounds-consistent *relative to C if the value $min\{D_x\}$ (respectively, $max\{D_x\}$) can be extended to a full tuple t of C. We say that t supports $min\{D_x\}$ (resp., $max\{D_x\}$). A constraint C is bounds-consistent if each of its variables is bounds-consistent.*

A constraint problem can be made bounds consistent by repeatedly removing unsupported lower and upper values from the domains of its variables. This level of consistency can be far more efficient to achieve for large domains and for global constraints.

Example 3.5.4 Consider the constraint problem with variables $x_1, ...x_6$, each with domains $1, ..., 6$, and constraints:

$$C_1 : x_4 \geq x_1 + 3, \quad C_2 : x_4 \geq x_2 + 3, \quad C_3 : x_5 \geq x_3 + 3, \quad C_4 : x_5 \geq x_4 + 1,$$

$$C_5 : alldifferent\{x_1, x_2, x_3, x_4, x_5\}$$

The constraints are not bounds consistent. For example, the minimum value 1 in the domain of x_4 does not have support in constraint C_1 as there is no corresponding value for x_1 that satisfies the constraint. Enforcing bounds consistency using constraints C_1 through C_4 reduces the domains of the variables as follows: $D_1 = \{1, 2\}$, $D_2 = \{1, 2\}$, $D_3 = \{1, 2, 3\}$ $D_4 = \{4, 5\}$ and $D_5 = \{5, 6\}$. Subsequently, enforcing bounds consistency using constraint C_5 further reduces the domain of C to $D_3 = \{3\}$. Now constraint C_3 is no longer bounds consistent. Reestablishing bounds consistency causes the domain of x_5 to be reduced to $\{6\}$. Is the resulting problem already arc-consistent? □

For the alldifferent constraints bounds-consistency can be enforced in $O(nlogn)$. The concepts of global constraints and bounds consistency were developed in the context of constraint programming languages and will be discussed again in Chapter 15.

3.6 Constraint propagation for numeric and Boolean constraints

When we process constraint systems that have a special syntactic form, such as numeric constraints expressed algebraically or boolean constraints expressed by boolean expressions,

the composition operation can be applied directly to the syntactic form of those expressions. We will demonstrate the implication of arc- and path-consistency, generalized arc-consistency, and relational arc-consistency on these specialized languages.

Algebraic constraints

Consider the following unary and binary numeric constraints restricted to the integers and defined over closed intervals:

$$D_x: \ x \in [1, 10], \quad D_y: \ y \in [5, 15], \quad R_{xy}: \ x + y = 10$$

Clearly x is not arc-consistent relative to y, nor is y arc-consistent relative to x. For example, $x = 10$ is in D_x but there is no value of y in D_y satisfying $x + y = 10$. Enforcing arc-consistency would restrict the domain of x to $x \in [1, 5]$ and the domain of y to $[5, 9]$. In this case, composition can be expressed as linear elimination; from $x + y = 10$ and $-y \leq -5$ we infer $x \leq 5$ by summing the first two expressions.

Now assume that we also have variable z with domain $D_z : z \in [-10, 10]$ and a constraint $R_{yz} : y + z \leq 3$. Clearly, x and z are not path-consistent relative to y since, for example, the pair $x = 1, z = -3$ cannot be consistently extended to a value of y. We can make x, z path-consistent relative to y by adding the constraint $x - z \geq 7$, which is obtained by summing $x + y = 10$ and $-y - z \geq -3$ (see also exercise 16).

Finally, as we already saw, if we have non-binary numeric constraints such as $x + y + z \leq 10$ over non-negative integer domains, applying generalized arc-consistency yields $x \leq 10$, $y \leq 10$, and $z \leq 10$. Enforcing relational-arc consistency requires additional constraints such as $x + y \leq 10$.

Boolean constraint propagation

Consider two clauses:

$$(A \vee \neg B) \text{ and } (B).$$

*(A) is equivalent to the proposition that "*A *is true" and $(\neg B)$ is equivalent to the proposition that "*B *is false." In this case, B is arc-consistent relative to A, but A is not arc-consistent relative to B. This is because, if A is false, B has to be false, but it is restricted to being true. Arc-consistency is achieved if we add the unary constraint (A), restricting A's domain to be true. In this boolean domain, the composition operation in Eq. (3.1) takes the form of* resolution*: from $(A \vee \neg B)$ and (B), we can infer (A). If we have, in addition, the clause $B \vee C$, then enforcing path-consistency relative to B requires the additional clause $A \vee C$. This clause can be obtained by resolution over B of the two clauses $(A \vee \neg B)$ and $(B \vee C)$.*

Procedure UNIT-PROPAGATION
Input: A CNF theory φ, $d = Q_1, ..., Q_n$.
Output: An equivalent theory such that no unit clause
appears in any non-unit clause.
1. queue = all unit clauses.
2. **while** queue is not empty, do:
3. $T \leftarrow$ next unit clause from Queue.
4. **for** every clause β containing T or $\neg T$
5. **if** β contains T delete β (subsumption elimination)
6. **else**, for each clause $\gamma = resolve(\beta, T)$,
if γ, the resolvent, is empty, the theory is unsatisfiable.
7. **else**, add the resolvent γ to the theory and delete β.
if γ is a unit clause, add to Queue.
8. **endfor**.
9. **endwhile**.

Figure 3.16: Algorithm unit propagation

Notice that applying generalized arc-consistency to non-binary clauses will not yield any domain reduction (a unit clause). Relational arc-consistency however, will require additional clauses to be deduced. Observe that since a domain constraint in cnf theories appears as a unit literal, the relational-arc consistency rule EQ. (3.3) translates to unit-resolution, that is, a resolution between a unit clause and an arbitrary clause. Indeed, UNIT PROPAGATION *is an algorithm equivalent to applying the rule in Eq. (3.3) until no new clauses can be deduced, at which point the set of clauses will satisfy relational-arc consistency. This yields a cost-effective inference algorithm for CNF theories that runs in linear time. Algorithm* UNIT-PROPAGATION *is given in Figure 3.16.*

Theorem 3.6.1 *Algorithm* UNIT-PROPAGATION *has a linear time complexity.*

Proof: *Each unit-resolution or unit-subsumption operation causes either the elimination of a clause or the shortening of a clause by one. Therefore the number of unit-resolutions is at most the length of the* cnf *formula.* □.

3.7 Trees, bi-valued networks and Horn theories

As we have seen, both arc and path-consistency can sometimes decide inconsistency. For some restricted classes of constraint problems, they are even guaranteed to solve the problem. Such tractable *classes will be discussed at length in later chapters. At this stage, we will only mention three such cases.*

Consider a binary constraint network whose graph is a tree, without cycles. If arc-consistency is enforced on such a tree network, and if no domain becomes empty as a result, then the problem is consistent (see exercise 11).

Next, consider the case of an arbitrary binary constraint network whose domain sizes are bounded to 2 (i.e., $k = 2$). If we apply path-consistency to such a bi-valued binary constraint network and no constraint becomes empty, then the network is guaranteed to have a solution. Moreover, a path-consistent bi-valued binary network is guaranteed to be minimal (exercise 7). An immediate implication of this observation for CNF theories is that if a CNF theory has clauses of length 1 or 2 only, then if we apply resolution (until nothing new can be generated) without deriving the empty clause, the theory is consistent.

Finally, assume a Horn CNF theory, that is, each clause has only negative literals or at most one single positive literal. If we apply unit propagation to such a Horn theory without generating the empty clause, then the theory is satisfiable. We summarize all these three cases in the next Theorem.

Theorem 3.7.1 *1. Binary constraint networks having no cycles can be decided by arc-consistency.*

2. *The consistency of binary constraint networks with bi-valued domains can be decided by path-consistency.*

3. *The consistency of Horn CNF theories can be decided by unit propagation.*

The above theorem illustrates two types of tractable classes. The first is characterized by the topological properties of the constraint graph (e.g., trees) while the second by restriction on the constraints themselves (bi-valued, Horn). These examples will be generalized in Chapter 11 and Chapters 8 and 9.

3.8 Summary

In this chapter we introduced the basic inference algorithms for constraint processing: arc, path and i-consistency. Figure 3.17 demonstrates schematically how arc, path, and

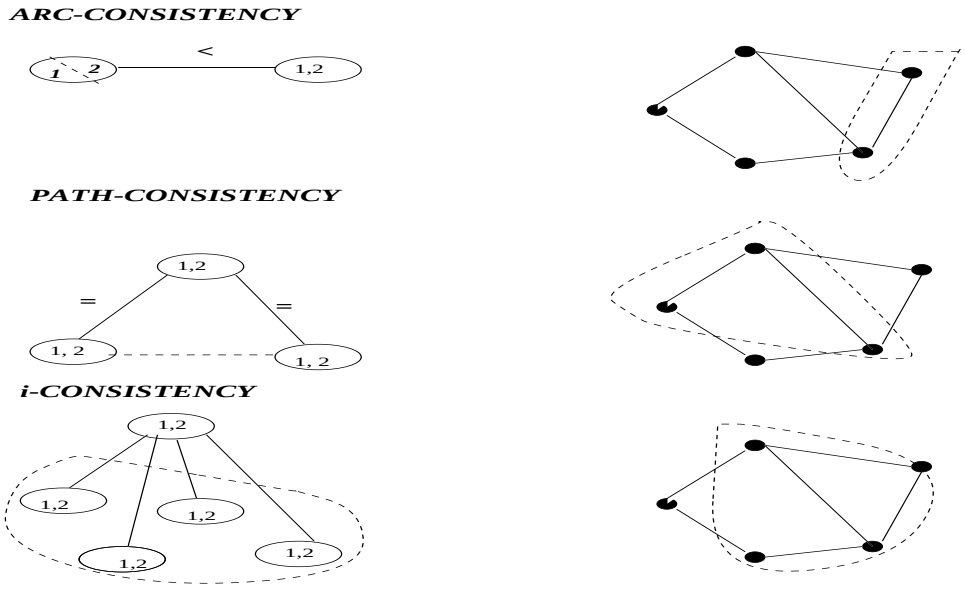

Figure 3.17: The scope of consistency enforcing: (a) arc-consistency, (b) path-consistency, (c) i-consistency

i-consistency are applied to one, two, or $i-1$ *variables. Arc-consistency applies to subnetworks of size 2, path-consistency to subnetworks of size 3, and* i*-consistency to subnetworks of size* i*.*

i-bounded consistency algorithms are polynomial (exponential in i*) and are not guaranteed to solve the problem. As* i *increases their ability of these algorithms to determine consistency also increases. Because consistency-enforcing algorithms infer new constraints, they change the structure of the network. Arc-consistency restricts the domains of variables, path-consistency restricts and add constraints on pairs of variables, and i-consistency enforces constraints of arity* $i-1$*. These changes can be reflected in the problem's constraint graph, as depicted in Figure 3.18. We further demonstrated the form of these consistency-enforcing methods for Boolean and numeric constraints. We also extended the basic constraint propagation algorithms to non-binary constraints via concepts such as generalized arc-consistency, relational arc-consistency, global constraints and bounds consistency. In Chapter 8 we will present another class of consistency algorithms that are defined by the constraints themselves.*

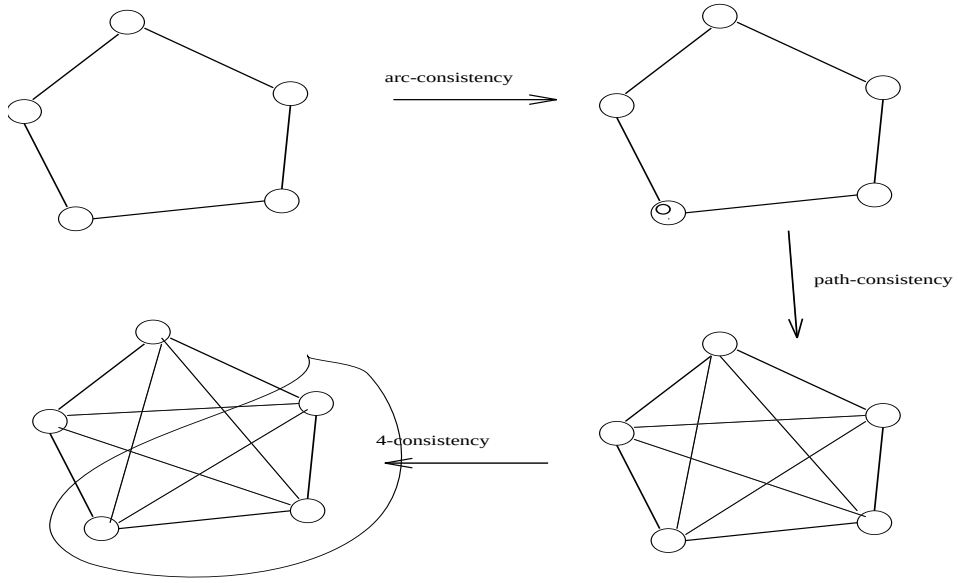

Figure 3.18: Changes in the network graph as a result of arc-consistency, path-consistency and 4-consistency.

3.9 Chapter notes

Work on constraint propagation and consistency algorithms was initiated by Waltz [291], who introduced a constraint propagation algorithm, later called arc-consistency, for three-dimensional cubic drawing interpretations. He demonstrated empirically that for some polyhedral figures, basic propagation algorithms are sufficient to solve these problems. This work was formalized and extended by Montanari [214], who also introduced the term "constraint networks." Montanari defined the notion of path-consistency and presented algorithms and analysis. Building upon these two works, Mackworth [195] distinguished three properties that characterize local consistency of binary networks (node, arc and path consistency) and also introduced several algorithms. He introduced AC-1, AC-2, and AC-3 for imposing arc-consistency, and PC-1, PC-2 and PC-3 for imposing path-consistency. The complexity of these algorithms were subsequently analyzed by Mackworth and Freuder [198], where they also observed that arc-consistency can solve tree-like problems. Mackworth also introduced generalized arc-consistency (also called hyper-arc consistency) non-binary constraints [196]. Freuder [95] then generalized these notions to k-consistency. Mohr and Henderson [212] presented optimal algorithms for arc and path-consistency, called AC-4 and PC-4, both of which were based on Laurier's earlier work on a language for solving combinatorial problems [184]. Mohr and Henderson's optimal algorithms were generalized for i-consistency in [51]. It was still observed that in practice the average

Figure 3.19: A constraint graph

performance of $AC-3$ is better than $AC-4$ [290]. Several authors have developed more efficient and refined arc-consistency algorithms for special classes of constraints, such as monotonic and functional constraints [83, 222]. Subsequent algorithms continue the numbering tradition (AC-5, AC-6, AC-7, PC-5, PC-6 and PC-7). Particularly noteworthy is AC-6 [24], which is able to achieve optimal worst-case complexity while avoiding some of the bad average case of AC-4 making it a serious competitor to both AC-4 and AC-3. Subsequent fine-tuning of AC-3 type algorithms towards being both worst-case optimal and average-case competitive, were presented by [26, 299].

Boolean constraint propagation was described by McAllester within the work of truth maintenance systems [205, 206]. Finally, in the context of constraint programming languages (see Chapter **??***) the specialized type of constraint propagation algorithms for global constraints were deveveloped for a variety of constraints. Most noteworthy are* alldifferent *constraints [237, 285], global cardinality constraint (gcc) that generalize the alldifferent constraint [238]. The* commulative constraint [6, 42] and the *cycle* constraint [20]. Bounds consistency was introduced in [231] and improved for the alldifferent constraint in [208].

3.10 Exercises

1. Consider a network having 8 variables named 1,2...,8, each having domains $\{1, 2, 3, 4\}$, whose constraints and graph are in Figure 3.19. Find an equivalent arc and path-consistent network. Is the path-consistent network minimal? Is it backtrack-free?

2. Consider the CSP formulation of the *Zebra problem*, where you have 25 variables, divided into clusters where the domains are house numbers (problem 5 in Chapter 2). The constraint graph of a possible formulation is given in Figure 3.20. Is your

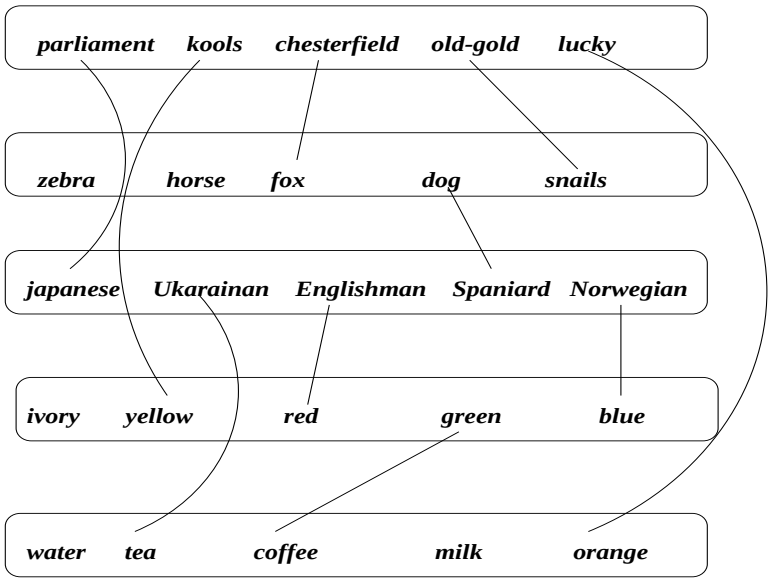

Figure 3.20: Constraint graph of the zebra problem

CSP formulation of the Zebra problem arc-consistent? If not, describe an equivalent arc-consistent network. Is it path-consistent?

3. Prove that algorithm *AC-1* generates an equivalent arc-consistent network. Prove also that when an empty domain is encountered, the network has no solution.

4. Prove that the time and space complexity of *AC-4* is $O(ek^2)$, where e is the number of constraints and k bounds the domain size.

5. Consider a binary network formulation of the crossword puzzle, and suppose that we are to fill it with the words from the following list, using each word not more than once:

   - HOSES, LASER, SAILS, SHEET, STEER,
   - HEEL, HIKE, KEEL, KNOT, LINE,
   - AFT, ALE, EEL, LEE, TIE.

   Is the problem arc-consistent? If not, find an equivalent arc-consistent network. Is the network path-consistent? If not, enforce path-consistency.

6. Prove that a minimal network is always path-consistent: any consistent pair of values can be consistently extended by any third variable.

7. Prove that a bi-valued, non-empty, path-consistent binary network is consistent. Prove that it is also a minimal network.

8. Implement algorithms AC-3 and AC-4.

   (a) Apply the algorithms to a set of randomly generated binary problems created using the 4 parameters N, K, C, T: N being the number of variables, K is the number of values in each domain, T is the tightness of each constraint (the number of allowed pairs divided by the total number of pairs), and C is the number of constraints. For each N, K C pairs of variables are selected uniformly at random, and for each T pairs of values for each constraints are selected randomly as allowed pairs. Which algorithm is better? AC-3 or AC-4?

   (b) Apply AC-3 and AC-4 to the n-queens problem.

9. Consider a graph coloring problem with 3 colors having 4 nodes x_1, x_2, x_3, x_4 where every two nodes are connected. Is the problem arc-consistent? Path-consistent? 4-consistent? If not, enforce strong 4-consistency. Can you make the problem 4-consistent by adding only binary constraints?

10. Let $\mathcal{R}$ be an arbitrary graph 3-coloring problem.
    1. Discuss what would be the effect of enforcing 2 and 3-consistency on $\mathcal{R}$.
    2. What would be the effect of enforcing $2-$, $3-$, $4-$, $...k-1-$, $k-$, $k+1-$ consistency on a k-coloring problem?

11. Prove that an arc-consistent binary network that has no cycle is consistent.

12. Derive a worst-case complexity analysis for algorithm *PC-3*.

13. Read algorithm PC-4 in [212] and show that the best-case performance of *PC-4* is $O(n^3k^3)$ or $O(n^3tk)$, where n is the number of variables, k bounds the domain sizes, and t bounds the number of tuples in a a relation.

14. Show that the two definitions of path-consistency (definitions 3.3.2 and 3.3.7) are equivalent only when *all* triplets of variables are considered.

15. A graph is *transitively closed* iff any two nodes connected by a path are directly connected. Prove that the constraint graph of a path-consistent network is transitively closed.

16. Generate an arc and path-consistent network equivalent to the network given by:

$$D_x : \ x \in [1, 10], \quad D_y : \ y \in [5, 15], \quad R_{xy} : \ x + y = 10$$

$$D_z : z \in [-10, 10], \quad R_{yz} : y + z \leq 3$$

Make the network generalized-arc-consistent and relational arc-consistent.

17. (due to J-C Regin) Consider the global alldiferent constraints over $\{x_1, ...x_{n/2}\}$ with domains [0..n] where the domains of $x_{n/2+1}, x_{n/2+2}$ is [1..2], $x_{n/2+3}, x_{n/2+4}$ have domains [4..5], $x_{n/2+5}, x_{n/2+6}$ have domains [7..8]. Make the global constraint bounds-consistent and analyze the complexity of your algorithm.

18. Derive the constraint alldifferent for the speaker problem presented in Section 3.5.2 and make this global constraint arc-consistent.

19. Prove Theorem 3.7.1 and analyze the complexity of deciding the consistency of binary tree networks, of bi-valued binary networks, and of Horn theories.